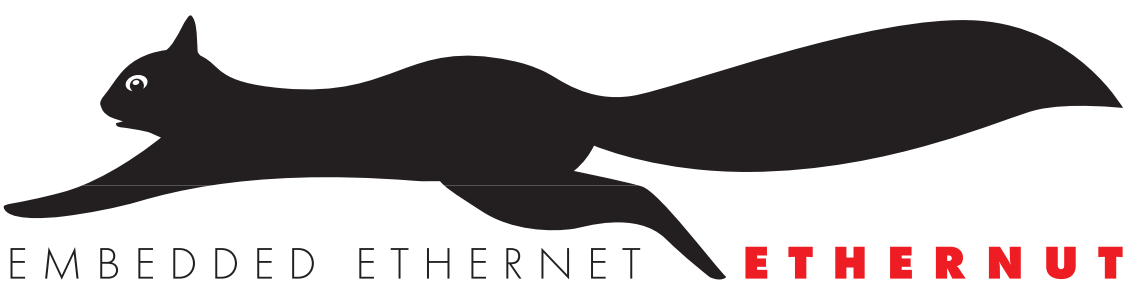
Ethernut

Software Manual



Manual Revision: 2.0
Issue date: September 2004

Copyright 2001-2004 by egnite Software GmbH. All rights reserved.

egnite makes no warranty for the use of its products and assumes no responsibility for any errors which may appear in this document nor does it make a commitment to update the information contained herein.
egnite products are not intended for use in medical, life saving or life sustaining applications.
egnite retains the right to make changes to these specifications at any time, without notice.
All product names referenced herein are trademarks of their respective companies. Ethernut is a registered trademark of egnite Software GmbH.

Contents

About Nut/OS and Nut/Net 1
- Nut/OS Features 1
- Nut/Net Features 1

Quick Start with ICCAVR 2
- Installing ICCAVR 2
- Installing Nut/OS 2
- Configuring Nut/OS 2
- Configuring ImageCraft 6
- Creating the First Nut/OS Application 9

Quick Start with AVR-GCC on Linux 12
- Installing AVR-GCC on Linux 12
- Installing Nut/OS 12
- Configuring Nut/OS 12
- Creating the First Nut/OS Application 17

Quick Start with WinAVR 19
- Installing AVR-GCC on Windows 19
- Installing Nut/OS 19
- Configuring Nut/OS 19
- Creating the First Nut/OS Application 23

Running the Embedded Webserver 25

Nut/OS 27
- System Initialization 27
- Thread Management 27
- Timer Management 28
- Heap Management 29
- Stream I/O 30
- File System 31
- Device Drivers 32

Nut/Net 33
- Network Device Initialization with DHCP 33
- Socket API 34
- Protocols 36
- Conversion Function 37
- Network Buffers 37

Simple TCP Server 39
- Initializing the Ethernet Device 39
- Connecting a Client to a Server 40
- Communicating with the Client 41
- Trying the Sample Code 42

Reference Material 43

Troubleshooting 45

About Nut/OS and Nut/Net

Connects embedded applications to a local Ethernet and the global Internet.

Nut/OS Features

Nut/OS is a very simple Realtime Operating System (RTOS) providing the following features

- Open Source
- Modular design
- Cooperative multithreading
- Event queues
- Dynamic memory management
- Timer support
- Stream I/O functions
- Expandable device driver interface
- File system support

Nut/Net Features

Nut/Net is a TCP/IP stack providing

- Open Source
- ARP, IP, UDP, ICMP and TCP protocol over Ethernet and PPP
- Automatic configuration via DHCP
- HTTP API with file system access and CGI functions
- TCP and UDP socket API for other protocols

The first decision, that has to be made, is to select the development platform you want to use, either ImageCraft's ICCAVR or GNU's AVR-GCC. The commercial ImageCraft Compiler offers an advanced IDE and is the first choice of most professional developers using a Windows PC. The GNU compiler is available for Linux and Windows.

The next chapters will guide you to quickly set up and start Nut/OS application development for each environment. It is assumed, that you are using Nut/OS 3.9.1 or above. Earlier releases do not differ much, but did not include the Configurator.

Quick Start with ICCAVR

Getting a professional environment up and running.

Users of GCC will skip this chapter.

A licensed version of the compiler is required to rebuild Nut/OS, the demo version will not work. A license is available from most distributors of the Ethernut Hardware or from ImageCraft directly.

Installing ICCAVR

Please follow the installation instructions that came with your compiler. By default the target directory will be C:\icc.

Installing Nut/OS

The installation for Windows is packed into two self-extracting executables, nutXYZc.exe and nutXYZd.exe, where XYZ has to be replaced with the version number. The first file contains complete code and tools and should be installed before second file, which contains the documentation.

By default all files will be installed in C:\ethernut\nut, which is called the Nut/OS top source directory.

Both installation files add a few entries to your Windows start menu, one of which is titled *Configure Development Environment*, which directly carries us to the next step.

Configuring Nut/OS

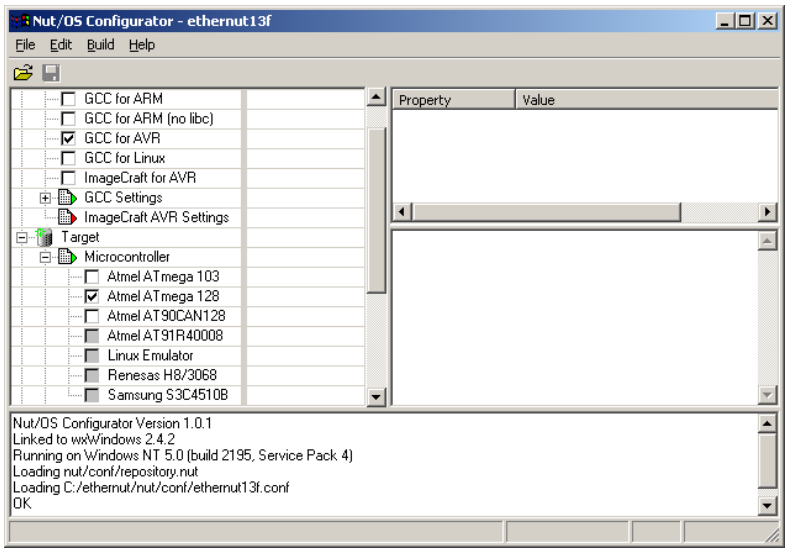
Select *Configure Development Environment* from the *Ethernut* entry in the *Windows Program Start Menu*. This will execute the Nut/OS Configurator. In this chapter we will not handle the details of this tool, but concentrate on the ICCAVR specific settings.

When started, a file selection dialog is presented. If not already displayed, navigate to the nut/conf directory, where the hardware configuration files are located.

Select the configuration file for your board and press Open.

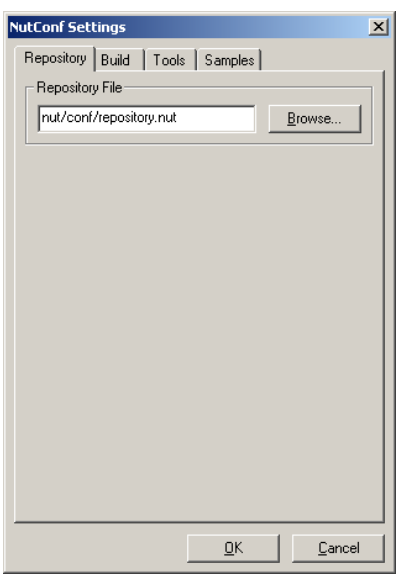
- ethernut13f.conf for board versions 1.3 Rev-F or previous.
- ethernut13g.conf for board versions 1.3 Rev-G.
- ethernut21b.conf for board versions 2.0 Rev-A or Ethernut 2.1 Rev-B.

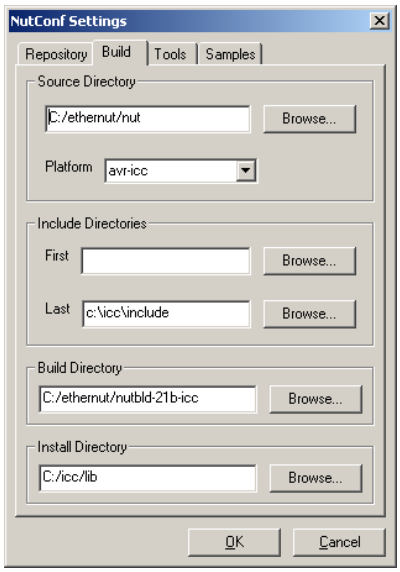
The selected file will be loaded and the hardware related configuration for this board will be automatically set by the Configurator.



Next the main configuration window will receive the focus. Select *Settings* from the *Edit* entry of the main menu. The settings notebook with four pages named *Repository*, *Build*, *Tools* and *Samples* will appear.

Usually the first page can be left unchanged. The entry specifies the path to the Nut/OS component repository.





On the second page enter the correct top source directory, where you installed Nut/OS. The Configurator will scan the conf subdirectory of the specified path for predefined platform settings and add them to the *Platform* drop down list.

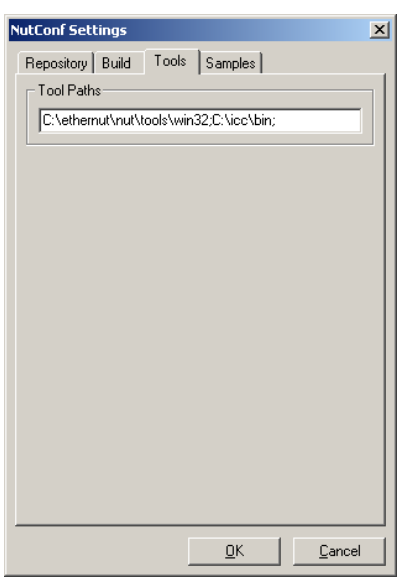
Select avr-icc as the platform.

The *First* include directory should be left empty. Enter the path of your ImageCraft include directory as the *Last* include directory. When building Nut/OS, the Configurator will automatically add all Nut/OS include directories.

It is important to understand, that we distinguish between the source directory and the build directory. The big advantage is, that we are able to create several system builds for different configurations from a single source tree, e.g. building Nut/OS for different Ethernut board revisions.

The build directory can be located anywhere, but it is recommended to put it in the parent of the top source directory and choose a meaningful name. Using the *Browse* button offers to create new directories.

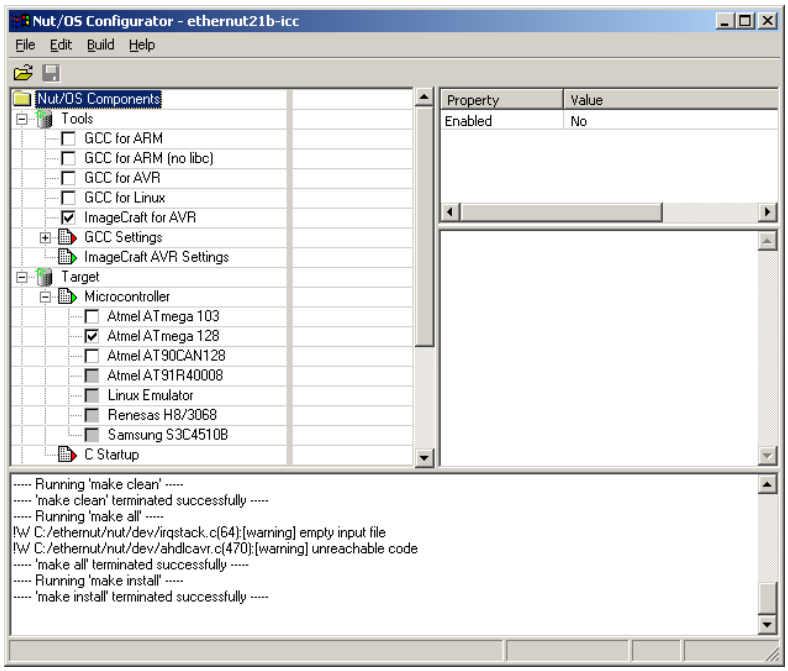
Finally enter an *Install Directory*. The ImageCraft compiler can't search more than one directory for libraries. Thus it is required to enter the path name of the ICCAVR lib directory. After system build, the Configurator will copy the libraries to this destination.



On the third page enter the paths to the directories containing the required tools, separated with semicoli. One additional semicolon is required after the last path. The Nut/OS tools should be first, followed by the ImageCraft bin directory.

The last page titled *Samples* can be left alone when using ICCAVR. The Nut/OS installation already created ready-to-use project files for the ImageCraft IDE.

Press OK to save your entries.



We are back to Configurator's main window with one configuration option left. Actually we already selected the proper tools when specifying the platform. Due to the internal structure used by the Configurator, we need to specify once again, that we are going to use the ImageCraft compiler. Click on *Tools* in the tree shown on the left to open the trunk and select *ImageCraft for AVR*.

Two final steps are required to build the configured Nut/OS.

First, select *Generate Build Tree* from the *Build* item of the main menu. After confirming the message box, the configuration tool prepares a new build directory or modifies an already existing one, using the build path specified in the settings notebook. It creates a set of files including some C language header files in a subdirectory named cfg. These files are included into the Nut/OS source code to tailor the system to a specific hardware.

Second, select *Build Nut/OS* from the *Build* item of the main menu. The configurator will run an external utility to clean any previous build (make clean), create new libraries (make all) and finally copy them to the install directory (make install).

The last step created the following ICCAVR libraries:

- libnutcrt.a (C runtime support library)
- libnutcrtf.a (C runtime support library with floating point)
- libnutdev.a (Device driver library)
- libnutfs.a (File system library)
- libnutnet.a (Network library)
- libnutos.a (RTOS library)
- libnutpro.a (Protocol library)
- crtnut.o (ImageCraft runtime initialization)
- crtnutram.o (ImageCraft runtime initialization with external RAM variables)
- nutinit.o (Nut/OS initialization)

If this step fails, check your settings again. If you can't determine the problem try the command line to fix this or at least find out more about what went wrong.

Open a DOS window, change to the build directory directory and enter

```
PATH=c:\ethernut\nut\tools\win32;c:\icc\bin;%PATH%
```

to add the directories containing all required tools to your PATH environment. Note, that the directories given above are an example. You may have installed ImageCraft and Nut/OS in different directories.

Enter

```
make clean
```

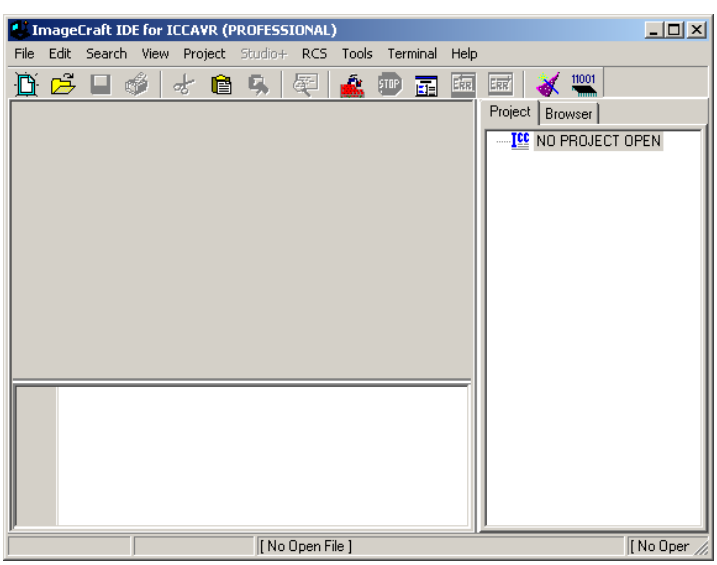
followed by

```
make install
```

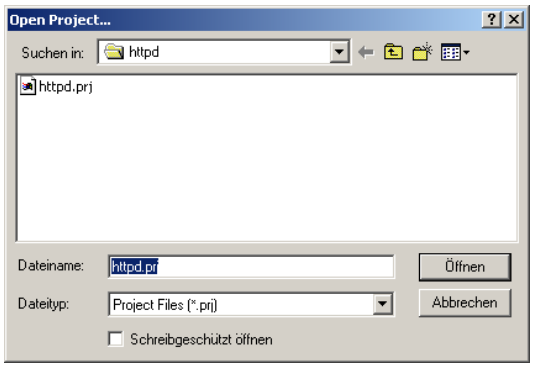
At the end you should see some kind of error message. If this is all Greek to you, best to ask the Ethernut mailing list for further assistance.

Configuring ImageCraft

Everything is ready to start the ImageCraft IDE and try our first Ethernut project. When started for the first time, the IDE will appear without any opened project.

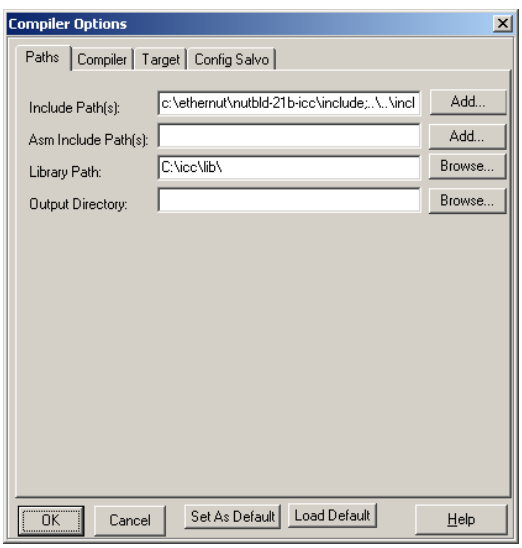


Select *Open...* from the project menu, navigate to the Nut/OS top source directory and load the project file httpd.prj, which is located in the subdirectory appicc/httpd.



The prepared webserver project contains most required settings. Later on, you will create your own projects and refer to the following steps to configure it.

If the project files in your Nut/OS distribution had been created for a different version of the ImageCraft compiler, you will see a warning message, reminding you to select the correct target. You can ignore this for now, we will get to this step anyway.



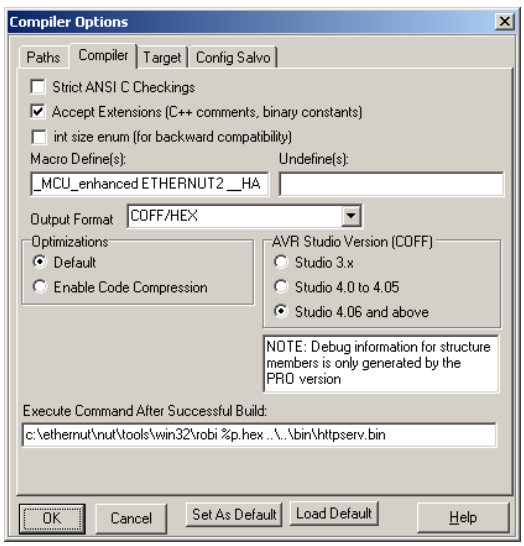
Select Options... from the project menu and click on the tab named Paths. As you can see in the picture, that two additional paths c:\ethernut\nutbld-21b-icc and ..\..\include\ had been added to the Include Path(s), so that ICCAVR can find all required include files.

The first path points to the include files created by the Configurator in the build directory. These files mainly contain hardware specific definitions.

The second path is for Nut/OS standard include files. It may contain files with the same name as the build include, but the compiler will ignore them, if they are already found in the first directory.

The third directory, C:\icc\include\ points to the standard include path of ICCAVR. If you installed your compiler in a different directory, then change this entry. The same applies to the *Library Path*.

Click on the *Compiler* tab and verify, that all options are correctly set. Code size reduction is optional and only available in the professional version of the ImageCraft compiler.



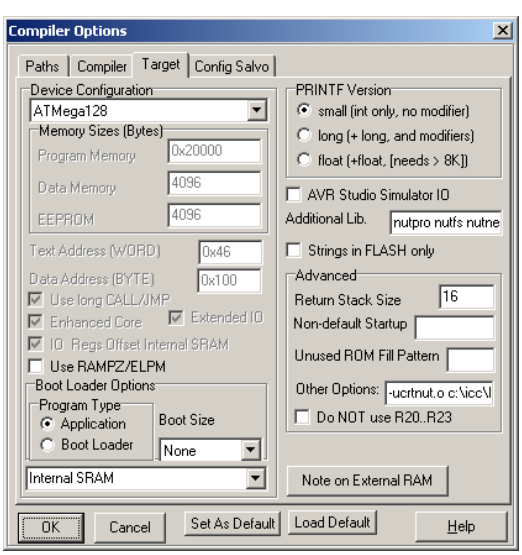
Add __MCU_enhanced to the macro definitions only, if you are compiling for the ATmega128 CPU.

Add ETHERNUT2 to the macro definitions only, if you are compiling for the Ethernut 2.0 or 2.1 board.

In any case you need to add __HARVARD_ARCH__. All Macro Defines should be separated by spaces.

The robi command entered for the *Execute Command After Successful Build* at the bottom can be used to convert a hex file to a raw binary and is not required. You may use raw binary files later, when replacing the JTAG or ISP adapter by the Ethernet bootloader. Loading programs into the Ethernut via a bootloader is much faster than any other method of in-system-programming.

Click on the *Target* tab. Again make sure, that the right options are set. Specifically check the device for ATmega103 or ATmega128.



The following *Additional Libs* are required for the webserver project:

```
nutpro nutfs nutnet nutos
nutdev nutcrt.
```

If your application requires floating point operations, replace nutcrt by nutcrtf. This will, however, produce larger code.

For Other Options enter

```
-ucrtnut.o
c:\icc\lib\nutinit.o
```

which will instruct the compiler to use a Nut/OS specific runtime initialization routine. The main difference to the standard routine is, that NutInit is called before main, so any RTOS specific initialization is hidden from your application code. You can simply start coding main, while the idle thread and the Nut/OS timer is already running in the background. In fact, main is started as a Nut/OS thread. This feature makes Nut/OS applications look like normal C programs and preserves portability.

Most Nut/OS applications will not require more RAM than 4 kBytes for their variables. Thus, the compiler is intentionally set to *Internal SRAM*. All external RAM is managed by the Nut/OS heap and dynamically allocated and released.

You can find more information about this topic in a separate document named Nut/OS Memory Considerations.

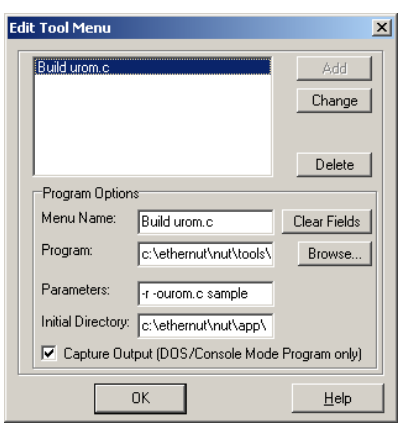
If you define large arrays or structures as pointers and allocate them with malloc from heap memory, your application will feel fine with this limitation. However, sometimes your variables may not fit. In these cases, select *External SRAM of 32 kByte* and replace crtnut.o by crtnutram.o in *Other Options*.

Recently a minor problem had been detected, which may be fatal for your application. If the compiler variables are near the limit of the internal 4 kBytes RAM, they may overlay with the idle thread's stack. Fortunately ICCAVR produces linker results in a map file (extension .mp). It is always a good idea checking this file. It will give you a general overview about memory usage. Check this file for __bss_end. This is the last RAM address used by your program variables. It should be below 1000 hex for the ATmega128 or 0F00 hex for the ATmega103. In most cases it helps to move your string literals to flash memory. But never enable the compiler option *Strings in FLASH only* with Nut/OS programs. In our view, this is a bad compiler option anyway. In Nut/OS you can use the type prog_char for these strings.

We ignore the last page for Salvo settings. Salvo is another RTOS similar to Nut/OS. Press OK to close the *Compiler Option Window*.

Specially for our HTTP Server project, a special entry in *Tools Menu* will be helpful. Select *Configure Tool* and add the following entries in the *Edit Tool Menu Dialog*.

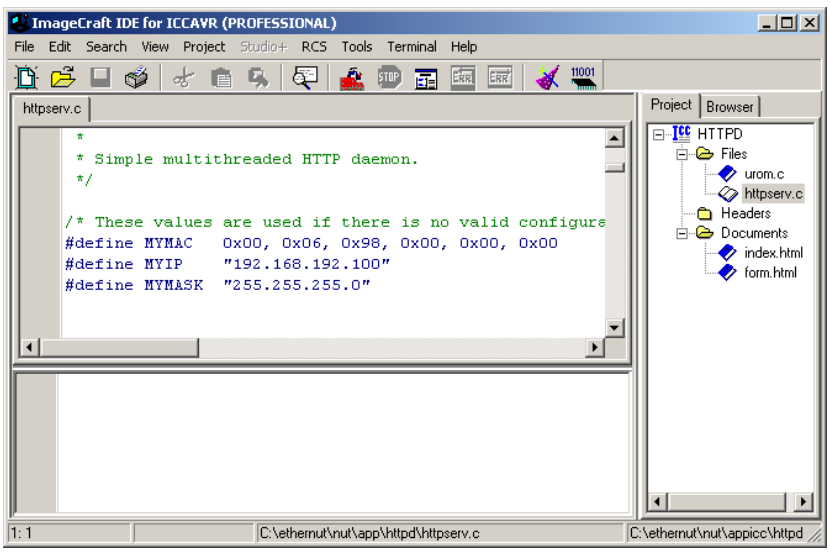
Menu Name: Build urom.c
Program: c:\ethernut\nut\tools\win32\crurom.exe
Parameters: -r -ourom.c sample
Initial Directory: c:\ethernut\nut\app\httpd
Activate Capture Output



The crurom utility converts all files in a directory to a C source file. This is used to include HTML files, images, Java Applets or other stuff into the Webserver's simple file system. The C source will be compiled and linked to the Nut/OS code.

Creating the First Nut/OS Application

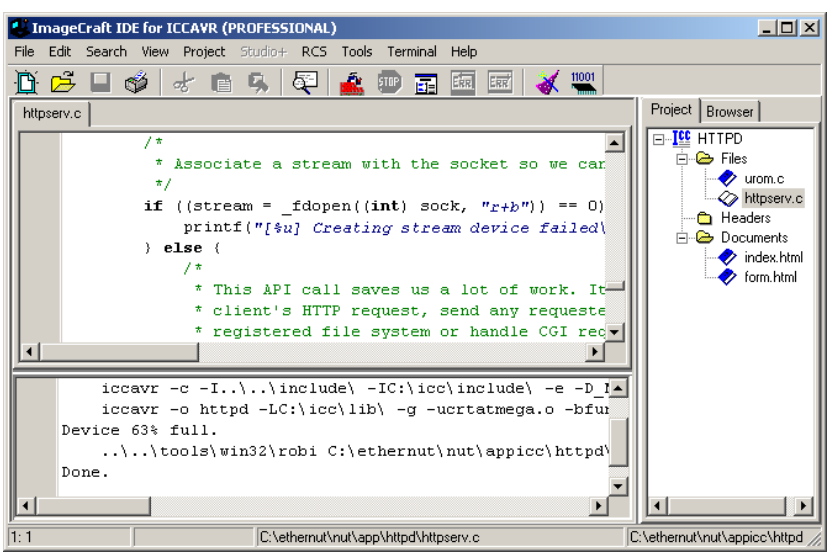
By default Nut/OS uses DHCP to automatically setup its TCP/IP configuration. Even without DHCP, typical applications will store these settings in the on-chip EEPROM. In order not to overload this tutorial, we will use hard coded addresses.



Open the file httpserv.c in the integrated editor. Consult your ICCAVR manual on how to do this and how to use the editor. You probably have to change the IP address and may also modify the IP mask to fit your network environment. Otherwise your web browser won't be able to talk to your Ethernut Board later on. If unsure what to do, better ask someone with IP network experience.

You can change the MAC Address to the one, which you received with your Ethernut Board. For self build boards or other boards without MAC address, you need at least to make sure, that the address is unique in your local network.

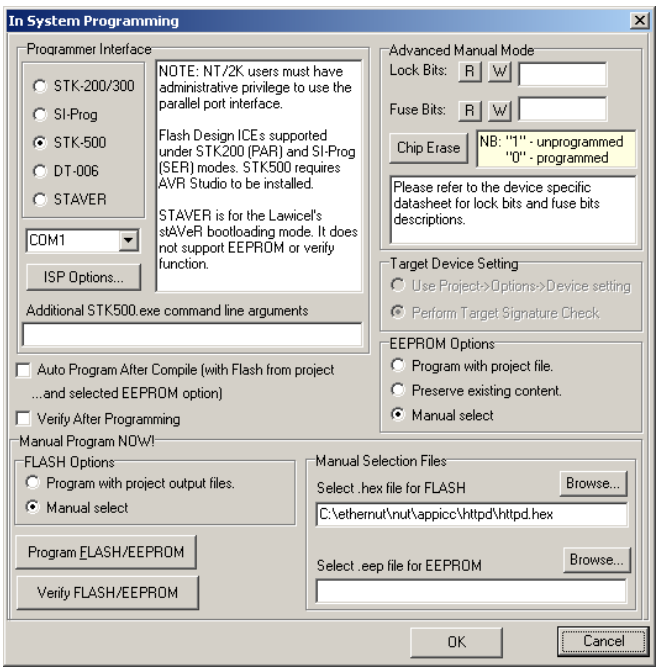
When selecting *Make Project* from the project menu, ICCAVR will compile and link the webserver code. Check, that no errors occurred during this process. Refer to the ICCAVR manual for further details.



As a result of the this step, ICCAVR created several files in the project directory. One is named httpserv.hex and contains the binary code in Intel Hex Format. The second file, httpserv.cof, can be loaded into the AVR Studio Debugger and contains the binary code plus additional debugging information.

Unless debugging is required, you can directly use the ICCAVR IDE to program the ATmega flash memory with an ISP Adapter.

If not already done, connect the ISP Adapter to the Ethernut Board and the PC. While doing this, you must have switched off the Ethernut's power supply.



Switch the Ethernut power supply back on and select *In System Programmer* from the *Tools* menu. Then select the file httpserv.hex for the *FLASH* by clicking on the *Browse* button. We do not need to program the EEPROM. Now press the button labeled Program FLASH/EEPROM to start in-system programming. This takes some seconds. Some version of ICCAVR display an error message on empty EEPROM entries. You can ignore this.

Finally press the OK button to close the *In System Programming* window. Your Ethernut Board will immediately start the webserver application, waiting for a web browser to connect.

Skip the next two chapters, which are related to GCC.

Quick Start with AVR-GCC on Linux

Using free tools on a free platform.

Installing AVR-GCC on Linux

The Windows version of the compiler will be presented in the next chapter.

The following packages are required to build and install an AVR development environment on Linux:

- GNU Binutils 2.14
- GNU Compiler Collection GCC 3.3.
- AVR Libc
- Uisp

Detailed information is available at
www.nongnu.org/avr-libc/user-manual/install_tools.html

Installing Nut/OS

The installation for Linux is packed into an RPM package. When running

```
rpm -i ethernut-X.Y-Z.i386.rpm
```

everything will be installed in /opt/ethernut/nut, which is the Nut/OS top source directory. Alternatively it is possible to unpack the compressed tar with

```
tar -xzvf ethernut-X.Y-Z.src.tar.gz
```

where XYZ has to be replaced by the version number.

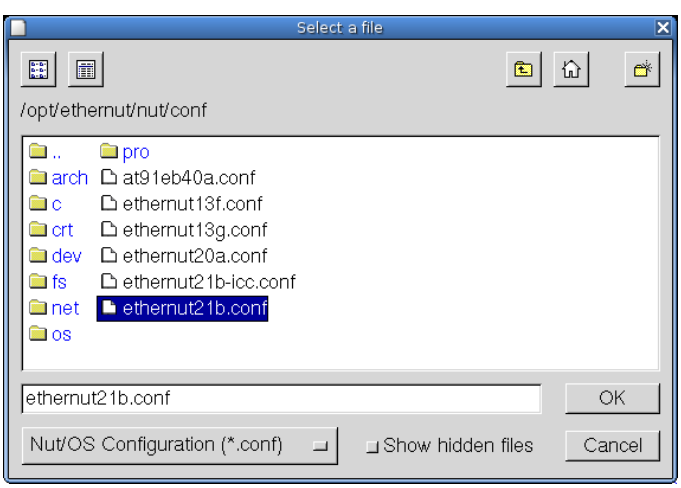
Configuring Nut/OS

It is assumed, that you are running a GTK based window manager. Change to the parent of the top source directory and enter

```
nut/tools/linux/nutconf
```

This will execute the Nut/OS Configurator. In this chapter we will not explain the details of this tool, but concentrate on the AVR-GCC specific settings.

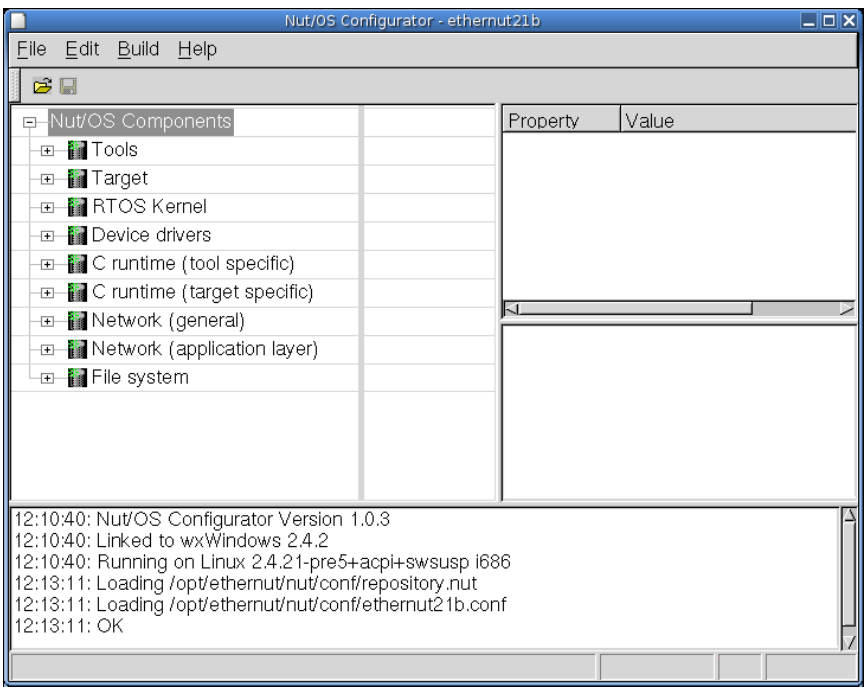
When started, a file selection dialog is presented. If not already displayed, navigate to the ethernut/nut/conf directory, where the hardware configuration files are located.



Select the configuration file for your board:

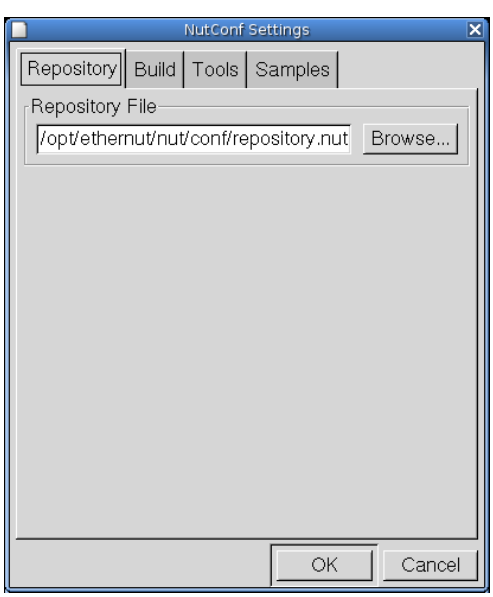
- ethernut13f.conf for board versions 1.3 Rev-F or previous.
- ethernut13g.conf for board versions 1.3 Rev-G.
- ethernut21b.conf for board versions 2.0 Rev-A or Ethernut 2.1 Rev-B.

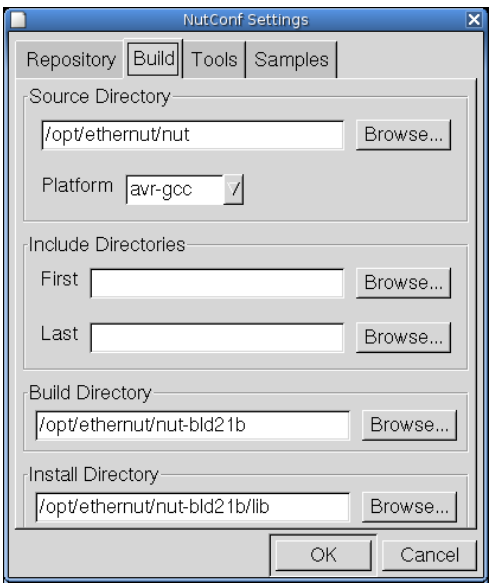
After pressing OK, the selected file will be loaded and the hardware related configuration for this board will be automatically set by the Configurator. The main configuration window will receive the focus.



From the *Edit* entry of the main menu select *Settings*. The settings notebook with four pages named *Repository*, *Build*, *Tools* and *Samples* will appear.

The first page defines the path the the so called repository file. You may want change the relative path to an absolute one.





On the second page enter the correct top source directory, /opt/ethernut/nut by default. The Configurator will scan the conf subdirectory in the specified path for predefined platform settings and add them to the *Platform* drop down list.

Select avr-gcc as the platform.

The entries for *First* and *Last* include directories can be ignored when using GCC. For special configurations they allow to set include directories, which will be searched before (*First*) or after (*Last*) the standard directories in the source tree.

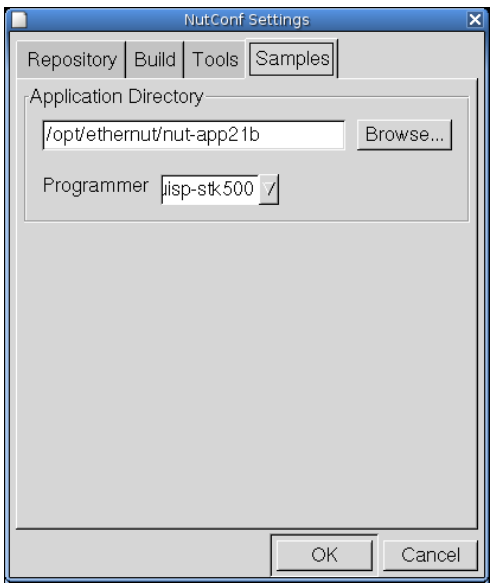
When building Nut/OS, the Configurator will automatically add all Nut/OS include directories.

It is important to note, that we distinguish between the source directory and the build directory. The big advantage is, that we are able to create several system builds for different configurations from a single source tree, e.g. building Nut/OS for different Ethernut board revisions.

The build directory can be located anywhere, but it is recommended to put it in the parent of the top source directory and choose a meaningful name. Using the *Browse* button offers to create new directories.

Finally enter an *Install Directory*. It is a good idea to choose a directory within the build tree, e.g. /opt/ethernut/nutbld-21b/lib. After system build, the Configurator will copy the libraries to this destination.

On the third page enter the paths to the directories containing the required tools, separated with semicoli. Do not miss to add a semicolon after the last path. The Nut/OS tools should be first, followed by any optional directory, which is not included in your PATH environment. The Configurator will add these entries to an existing PATH environment when running the compiler.



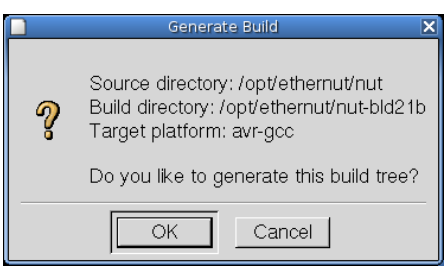
Entries on the last page of the settings notebook are used to create a sample application directory. Nut/OS comes with a few samples, which demonstrate its capabilities. This includes the HTTP server we are going to build in this chapter. The configurator can create a new directory for you, copying the sources from the source tree and creating templates for the Makefiles. Such a directory will be also quite usefull when writing your own Nut/OS applications later.

When selecting the correct *Programmer* from the drop down list, applications may be build and uploaded to the target board in one go with 'make burn'.

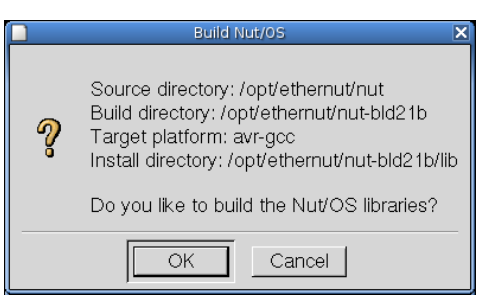
Press OK to save your entries.

The focus returns to the Configurator's main window. We are almost done, with one configuration option left. Actually we already selected the proper tools when specifying the platform. But due to the internal structures used by the Configurator, we need to specify once more, that we are going to use GCC for the AVR. Click on *Tools* in the tree on the left side of the main window to open the trunk and activate *GCC for AVR*, if not already selected.

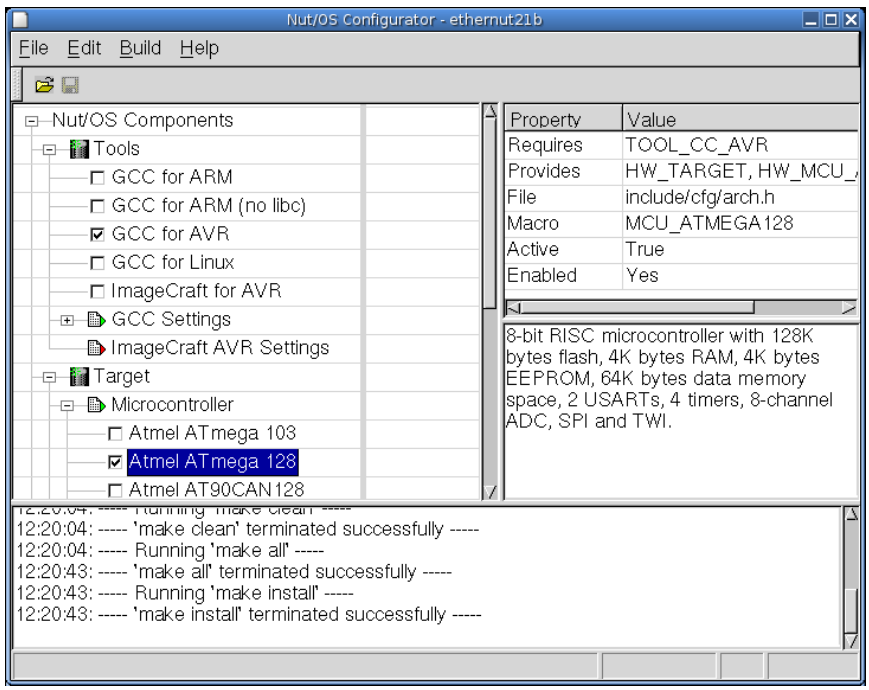
We are ready to build the configured Nut/OS, which is done in two steps.



First select *Generate Build Tree* from the *Build* entry of the main menu. After confirming the message box, the configuration tool prepares a new build directory or modifies an already existing one, using the build path specified in the settings notebook. It creates a set of files including some C language header files in a subdirectory named include/cfg. These files are included into the Nut/OS source code to tailor the system to our specific target hardware.



Then select *Build Nut/OS* from the *Build* entry of the main menu. After confirming the message box, the Configurator will run make clean, make all and finally make install.



The Configurator created the following AVR libraries:

- libnutcrt.a (C runtime support library)
- libnutcrtf.a (C runtime support library with floating point)
- libnutdev.a (Device driver library)
- libnutfs.a (File system library)
- libnutnet.a (Network library)
- libnutos.a (RTOS kernel library)
- libnutpro.a (Protocol library)
- nutinit.o (RTOS initialization)

If this step fails, check your settings again. If you can't determine the problem, then try the command line to fix this or at least find out more about what went wrong. Change to the build directory directory and enter

```
export PATH=/opt/ethernut/nut/tools/linux:$PATH
```

to add the directories containing the Nut/OS tools to your PATH environment. To build the libraries enter

```
make clean
```

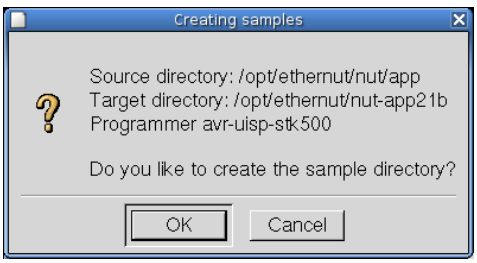
followed by

```
make install
```

At the end you should see some kind of error message. Best check the Ethernut FAQ or ask the Ethernut mailing list for further assistance.

Creating the First Nut/OS Application

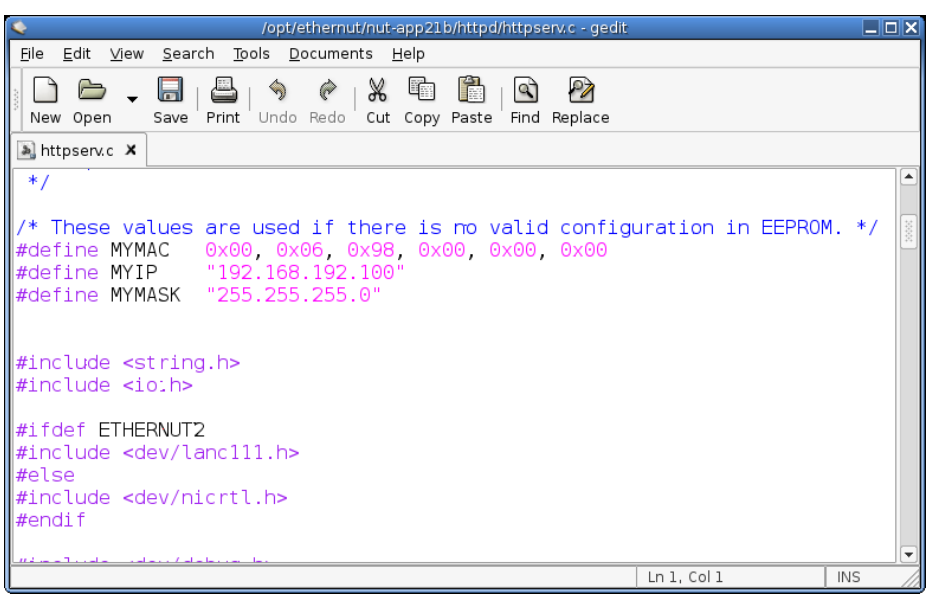
Although possible, we will not modify the source directory, which contains the application samples. Instead we will use the Configurator to create a copy of the sample directory for us.



Select *Create Sample Directory* from the *Build* entry of the main menu. After confirming the message box, the Configurator will create a new or update an existing directory for application development and fill in a copy of the Nut/OS sample applications.

By default Nut/OS uses DHCP to automatically setup its TCP/IP configuration. Even without DHCP, typical applications will store these settings in the on-chip EEPROM and thus will not require modifications of the source code. In order not to overload this tutorial, we use hard coded addresses.

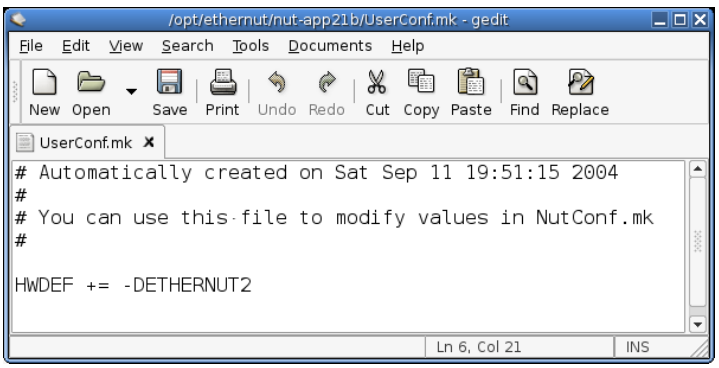
Run your favorite editor and load the file httpd/httpserv.c, which is located in the sample directory just created. You probably have to change the IP address and may also modify the IP mask to fit your network environment. Otherwise your web browser won't be able to talk to your Ethernut Board later on. If unsure what to do, better ask someone with IP network experience.



IYou can also change the MAC Address to the one, which you received with your Ethernut Board. For self build boards or other boards without MAC address, you need at least to make sure, that the address is unique in your local network. Save the changes.

When the Configurator created the sample directory, it added a user configuration file with default settings for Ethernut 2 boards, or, more precisely, for boards with the SMSC LAN91C111 Ethernet controller. If your board is equipped with a different controller (e.g. Charon II or Ethernut 1), you have to modify a second file in the sample directory, named UserConf.mk. Load this file in your editor and remove the entry

```
HWDEF += -DETHERNUT2
```



If you are creating the sample directory again using the same path, then the Configurator will overwrite all changes, but it will never modify UserConf.mk.

We are ready to build the webserver application and program the resulting binary into the target board.

Change to the httpd subdirectory within the sample directory and enter

```
make clean
```

followed by

```
make all
```

Check, that no errors occurred during this process. As a result, you will find the newly created binary file named httpd.hex in Intel hex format.

If not already done, connect your programming adapter to the Ethernut board and the proper PC interface and power up the Ethernut. In case you selected the correct adapter in the Configurator's settings dialog, you can simple enter

```
make burn
```

to upload the hex file to the Ethernut. Otherwise consult the documentation of your AVR programming software.

Quick Start With WinAVR

Running free tools out of the box.

Installing AVR-GCC on Windows

WinAVR (pronounced "whenever") is a suite of executable, open source software development tools for the AVR hosted on the Windows platform. They are mainly based on the GCC for AVR toolchain and are quite similar to the Linux tools. Detailed information is available at winavr.sourceforge.net.

Installing Nut/OS

The installation for Windows is packed into two self-extracting executables, nutXYZc.exe and nutXYZd.exe, where XYZ has to be replaced with the version number. The first file contains complete code and tools and should be installed first, while the documentation is packed in the second file.

By default all files will be installed in C:\ethernut\nut, which is called the Nut/OS top source directory.

Both installations files add a few entries to your Windows start menu, one of which is titled *Configure Development Environment*, which directly carries us to the next step.

Configuring Nut/OS

Select *Configure Development Environment* from the *Ethernut* entry in the *Windows Program Start Menu*. This will execute the Nut/OS Configurator. In this chapter we will not go to the details of this tool, but concentrate on the AVR-GCC specific settings.

When started, a file selection dialog is presented. If not already displayed, navigate to the nut/conf directory, where the hardware configuration files are located.

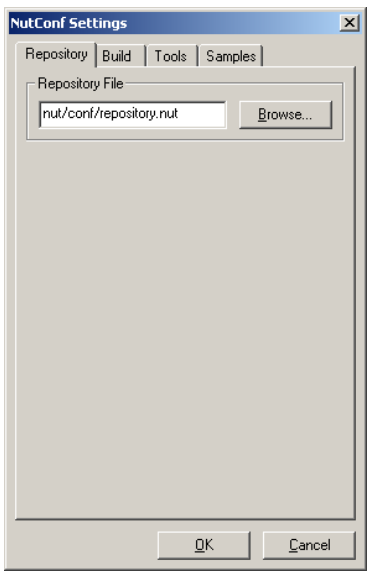
Select the configuration file for your board and press Open.

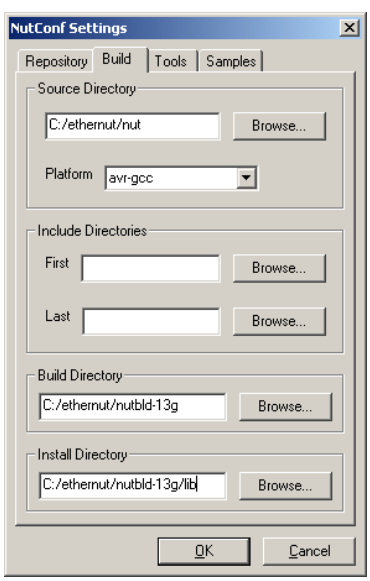
- ethernut13f.conf for board versions 1.3 Rev-F or previous.
- ethernut13g.conf for board versions 1.3 Rev-G.
- ethernut21b.conf for board versions 2.0 Rev-A or Ethernut 2.1 Rev-B.

The selected file will be loaded and the hardware related configuration for this board will be automatically set by the Configurator.

When the main configuration window receives the focus, select *Settings* from the *Edit* entry of the main menu. The settings notebook with four pages named *Repository*, *Build*, *Tools* and *Samples* will appear.

Usually the first page can be left unchanged.





On the second page enter the correct top source directory, where you installed Nut/OS. The Configurator will scan the conf subdirectory of the specified path for predefined platform settings and add them to the *Platform* drop down list.

Select avr-gcc as the platform.

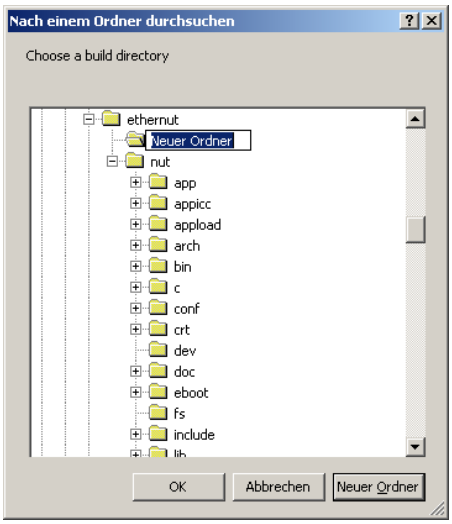
The entries for *First* and *Last* include directories can be ignored when using GCC. For special configurations they allow to set include directories, which will be searched before (*First*) or after (*Last*) the standard directories in the source tree.

When building Nut/OS, the Configurator will automatically add all Nut/OS include directories.

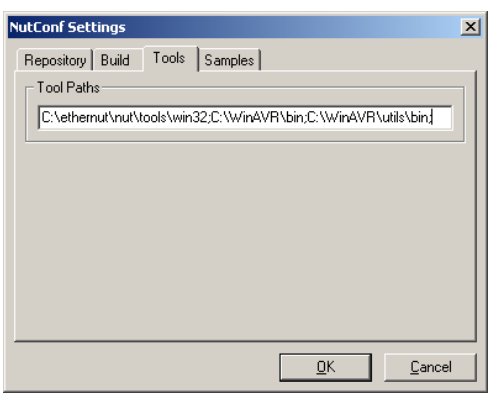
It is important to note, that we distinguish between the source directory and the build directory. The big advantage is, that we are able to create several system builds for different configurations from a single source tree, e.g. building Nut/OS for different Ethernut board revisions.

The build directory can be located anywhere, but it is recommended to put it in the parent of the top source directory and choose a meaningful name. Using the *Browse* button offers to create new directories.

Finally enter an *Install Directory*. It is recommended to choose a directory within the build tree, e.g. c:/ethernut/nutbld-21b/lib. After system build, the Configurator will copy the libraries to this destination.



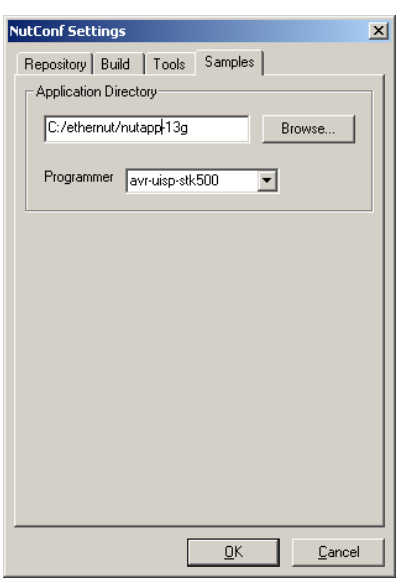
On the third page enter the paths to the directories containing the required tools, separated with semicoli. Do not miss to add a semicolon after the last path. The Nut/OS tools should be first, followed by the paths to the WinAVR executables (bin and utils/bin), if they are not included in your standard PATH environment. The Configurator will add these entries to the PATH environment when running the compiler.



Entries on the last page of the settings notebook are used to create a sample application directory. Nut/OS comes with a few samples, which demonstrate its capabilities. This includes the HTTP server we are going to build in this chapter. The configurator can create a new directory for you, copying the sources from the source tree and creating templates for the Makefiles. Such a directory may also host your own Nut/OS applications later.

It is quite useful to select a programmer from the drop down list, because applications may be build and uploaded to the target board in one go with make burn.

Press OK to save your entries.



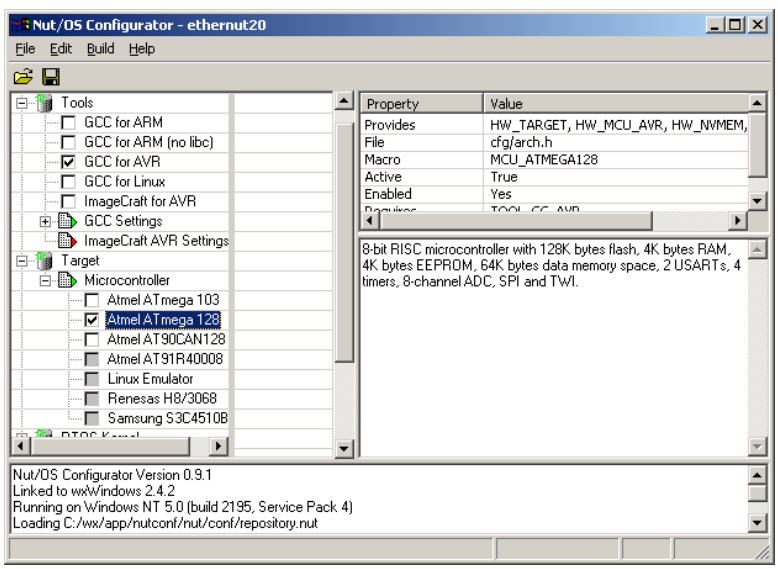
The focus returns to the Configurator's main window. We are almost done, with just one configuration option left. Actually we already selected the proper tools when specifying the platform. Due to the internal structures used by the Configurator, we need to specify once again, that we are going to use GCC for the AVR. Click on *Tools* in the tree on the left side of

the main window to open the trunk and activate *GCC for AVR*, if not already selected.

Build the configured Nut/OS is done in two steps.

First select *Generate Build Tree* from the *Build* entry of the main menu. After confirming the message box, the configuration tool prepares a new build directory or modifies an already existing one, using the build path specified in the settings notebook. It creates a set of files including some C language header files in a subdirectory named include/cfg. These files are included into the Nut/OS source code to tailor the system to our specific target hardware.

Then select *Build Nut/OS* from the *Build* entry of the main menu. The Configurator will now run make clean, make all and finally make install.



This step created the following AVR libraries:

- libnutcrt.a (C runtime support library)
- libnutcrtf.a (C runtime support library with floating point)
- libnutdev.a (Device driver library)
- libnutfs.a (File system library)
- libnutnet.a (Network library)
- libnutos.a (RTOS kernel library)
- libnutpro.a (Protocol library)
- nutinit.o (RTOS initialization)

If this fails, check your settings again. If you can't determine the problem try the command line to fix this or at least find out more about what went wrong:

Open a DOS command line window, change to the build directory directory and enter

```
set PATH=c:\ethernut\nut\tools\win32;%PATH%
```

to add the directories containing the Nut/OS tools to your PATH environment. To manually build the libraries enter

```
make clean
```

followed by

```
make install
```

At the end you should see some kind of error message. Best check the Ethernut FAQ or ask the Ethernut mailing list for further assistance.

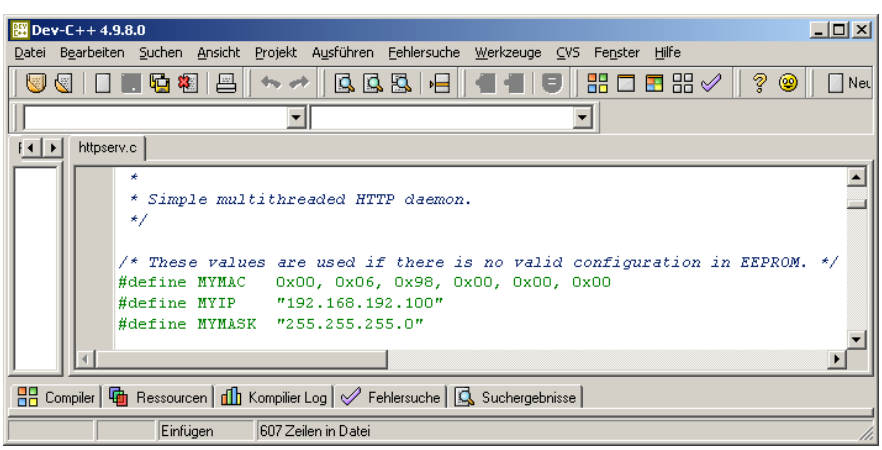
Creating the First Nut/OS Application

Although possible, we will not modify the source directory, which contains the application samples. Instead we will use the Configurator to create a copy of the sample directory for us.

Select *Create Sample Directory* from the *Build* entry of the main menu. After confirming the message box, the Configurator will create a new or update an existing directory for application development and fill in a copy of the Nut/OS sample applications.

By default Nut/OS uses DHCP to automatically setup its TCP/IP configuration. Even without DHCP, typical applications will store these settings in the on-chip EEPROM and thus will not require modifications of the source code. In order not to overload this tutorial, we use hard coded addresses.

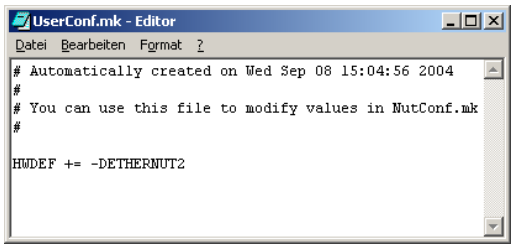
Run your favorite editor and load the file httpd/httpserv.c, which is located in the sample directory just created. You probably have to change the IP address and may also modify the IP mask to fit your network environment. Otherwise your web browser won't be able to talk to your Ethernut Board later on. If unsure what to do, better ask someone with IP network experience.



You may also change the MAC Address to the one, which you received with your Ethernut Board. For self build boards or other boards without MAC address, you need at least to make sure, that the address is unique in your local network. Save the changes.

When the Configurator created the sample directory, it added a user configuration file with default settings for Ethernut 2 boards, or, more precisely, for boards with the SMSC LAN91C111 Ethernet controller. If your board is equipped with a different controller (e.g. Charon II or Ethernut 1), you have to modify a second file in the sample directory, named UserConf.mk. Load this file in your editor and remove the entry

```
HWDEF += -DETHERNUT2
```



If you are creating the sample directory again using the same path, then the Configurator will overwrite all changes, but it will never modify UserConf.mk.

We are ready to build the webserver application and program the resulting binary into the target board.

Change to the httpd subdirectory within the sample directory and enter

```
make clean
```

followed by

```
make all
```

Check, that no errors occurred during this process. As a result, we will find the newly created binary named httpd.hex in Intel hex file format.

If not already done, connect your programming adapter to the Ethernut board and the proper PC interface and power up the Ethernut. In case you selected the correct adapter in the Configurator's settings dialog, you can simple enter

```
make burn
```

to upload the hex file to the Ethernut. Otherwise consult the documentation of your AVR programming software.

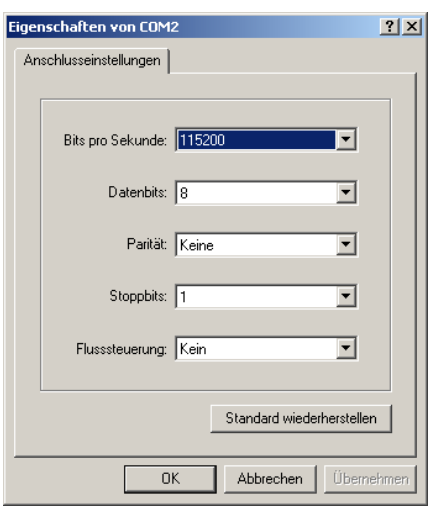
Running the Embedded Webserver

Connecting the 8-bit world.

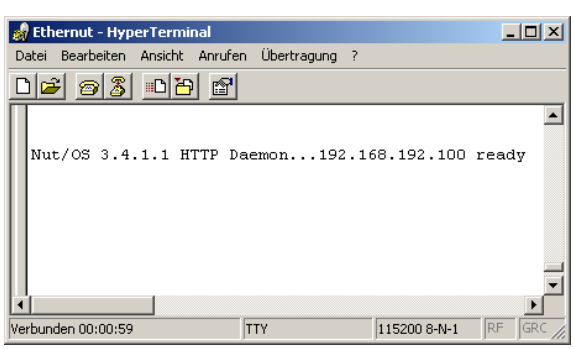
Although this chapter looks quite 'window-ish', it's targeted to Linux users too. In the previous chapters we compiled the binary for our first Nut/OS application, an embedded webserver, and uploaded it to the Ethernut board.

Most application samples use the serial port of the Ethernut board to provide some feedback about program progress or any kind of errors that may occur. Connect the serial port of your Ethernut with one of the serial ports on your PC using the cable that came with your Ethernut Starter Kit or any 1:1 DB-9 cable. There's no need to switch off the Ethernut, serial ports are quite safe and protected against shortcuts or electrical discharge. But remember, not to touch any bare contacts on the Ethernut board before taking some pre-cautions. Dissipate static electricity by touching a grounded metal object.

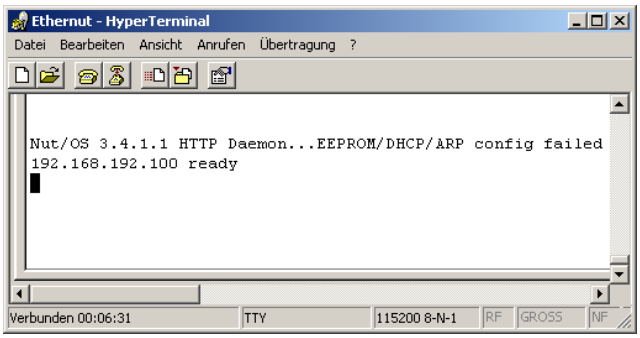
Now start your favorite terminal emulator on the PC or use the standard Hypertem on Windows. The required settings for the serial port are 115200 Baud, eight data bits, no parity, one stop bit and no handshake.



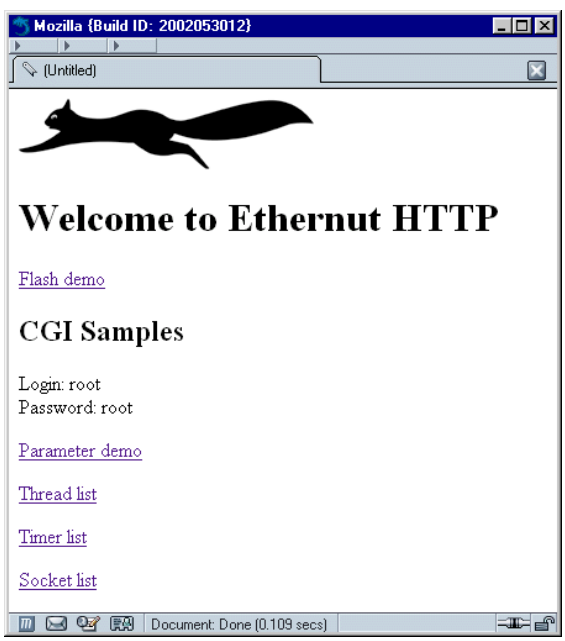
Make sure, that the Ethernut Board is connected to your local network. Resetting your Ethernut by pressing the reset button on the board will produce some text output in the terminal emulator's window.



The text on your system will differ, depending on your network configuration. If your network doesn't use DHCP you may get something like this:



Start a web browser on your PC. The URL to request is the IP address, which has been printed on the terminal emulator. If everything went well, you will see the main index page of the Ethernut webserver.



Congratulations, your embedded webserver is working!

The following chapters will introduce Nut/OS and Nut/Net in more detail. Some of you may not be able to follow every part. Don't worry. Try to code a few basic samples and have a look to the list of books at the end of this manual.

Nut/OS

A kernel routine overview.

Check the Nut/OS API Reference for a detailed description.

Be aware, that this chapter makes no attempts to explain details of the C language. It is assumed that you have a basic knowledge of C programming.

System Initialization

By default, C programs are started with a routine called main. This isn't much different in Nut/OS, however, the operating system requires certain initialization before the application is started. This initialization is included in a module named nutinit. It will initialize memory management and the thread system and start an idle thread, which in turn initializes the timer functions. Finally the application main routine is called. Because there's nothing to return to, this routine should never do so.

A sample application called *simple* demonstrates the most simple application, that could be build with Nut/OS. It does nothing else than running in an endless loop, consuming CPU time.

```
#include <compiler.h>
int main(void)
{
    for (;;);
}
```

The file compiler.h is included to fix a problem with AVR-GCC, which insists on setting the stack pointer again on entry to main(). When using this compiler, main is re-defined to NutAppMain. Other compilers won't be hurt.

Almost all Nut/OS header files do include compiler.h. Thus it is required only when no other header files are used in the application code.

Thread Management

Typically Nut/OS is at its most useful where there are several concurrent tasks that need to be undertaken at the same time. To support this requirement, Nut/OS offers some kind of light processes called threads. In this context a thread is a sequence of executing software that can be considered to be logically independent from other software that is running on the same CPU.

All threads are executing in the same address space using the same hardware resources, which significantly reduces task switching overhead. Therefore it is important to stop them from causing each other problems. This is particularly an issue where two or more threads need to share a resources like memory locations or peripheral devices.

The system works on the principle that the most urgent thread always runs. One exception to this is when a CPU interrupt arrives and the interrupt has not been disabled.

Each thread has a priority which is used to determine how urgent it is. This priority ranges from 0 to 254, with the lowest value indicating the most urgent.

Nut/OS implements cooperative multithreading. That means, that threads are not bound to a fixed time slice. Unless they are waiting for specific event or explicitly

yielding the CPU, they can rely on not being stopped unexpectedly. However, they may be interrupted by hardware interrupt signals. In opposite to pre-emptive multithreading, cooperative multithreading simplifies resource sharing and usually results in faster and smaller code.

As stated earlier, the main application thread is already running as a thread, together with the background idle thread, which is not visible to the application.

Creating a new thread is done by calling NutThreadCreate. The code running as a thread is nothing else than another C routine. To hide platform specific details, applications should use the THREAD macro to declare those routines.

```
THREAD(Thread1, arg)
{
    for (;;) {
        NutSleep(125);
    }
}

int main(void)
{
    NutThreadCreate("t1", Thread1, 0, 192);
    for (;;) {
        NutSleep(125);
    }
}
```

In this example the main thread creates a new thread before entering and endless loop. The new thread will run in a similar senseless loop.

It is important to keep the cooperative nature of Nut/OS in mind. NutSleep, which will be explained next, stops execution for a specified number of milliseconds. If one of the loops would not call this or any other blocking function like reading from a device or waiting for an event, then the other threads would never gain CPU control.

Timer Management

Nut/OS provides time related services, allowing application to delay itself for an integral number of system clock ticks by calling NutSleep(). A clock tick occurs every 62.5 ms by default, but may vary depending on the configuration.

The following minimal application will run in an endless loop, but spend most of the time sleeping for 125 milliseconds (two system clock ticks). During sleep time, any other thread may take over. However, if no other thread is ready to run or, as in our example, no other threads had been created, CPU control is passed to the Nut/OS Idle Thread.

```
#include <sys/timer.h>
int main(void)
{
    for (;;) {
        NutSleep(125);
    }
}
```

Another useful routine is NutGetCpuClock, which returns the CPU clock in Hertz.

Beside these directly called API routines, device timeouts are handled by the timer management. The following code fragment opens device “uart0” for binary reading and writing and sets the read timeout to 1000 milliseconds.

```
      int com;
      unsigned long tmo = 1000;
....
      com = _open("uart0", _O_RDWR | _O_BINARY);
      _ioctl(com, UART_SETREADTIMEOUT, &tmo);
```

Note, that Nut/OS uses on-chip hardware timer 0 of the ATmega CPU. All remaining timers are available for the application. You can use them in cases where the system timer resolution is insufficient.

Heap Management

Dynamic memory allocations are made from the heap. The heap is a global resource containing all of the free memory in the system. The heap is handled as a linked list of unused blocks of memory, the so called free-list.

Applications can use standard C calls to allocate and release memory blocks. Allocating a buffer for 1024 bytes, senseless filling it with 0xFF and releasing the buffer again, will look like this.

```
char *buffer;
buffer = malloc(1024);
if (buffer) {
     memset(buffer, 0xFF, 1024);
     free(buffer);
}
else {
     puts(“Out of memory error!”);
}
```

You should make intensive use of dynamic memory allocation for two reasons:

First, large local variables will occupy stack space. As each thread gets its own stack, you may waste a lot of memory by reserving large stacks.

Second, large global variables occupy space during the complete lifetime of the application, although they may not be used during that time. For some environments the global variable space may be much more limited than heap space.

Some developers argue, that dynamic memory allocation produces less reliable code and is slower. If you agree to this opinion, allocate all memory once during initialization.

The heap manager uses best fit, address ordered algorithm to keep the free-list as defragmented as possible. This strategy is intended to ensure that more useful allocations can be made. We end up with relatively few large free blocks rather than lots of small ones.

Event Management

Threads may wait for events from other threads or interrupts or may post or broadcast events to other threads.

For using events, a modified version of our thread example will look like this.

```
HANDLE evt_h;

THREAD(Thread1, arg)
{
    for (;;) {
        NutSleep(125);
        NutPostEvent(&evt_h);
    }
}

int main(void)
{
    NutThreadCreate("t1", Thread1, 0, 192);
    for (;;) {
        NutEventWait(&evt_h, NUT_WAIT_INFINITE);
    }
}
```

Again Thread1 is running an endless loop of sleeps. The main thread waits for an event posted to an event queue. Event queues are specified by variables of type HANDLE. The endless loop of the main thread is blocked in NutEventWait() and woken up each time an event is posted by Thread1.

Btw. it hadn't been very smart to introduce the variable type HANDLE. But it's there since the very early releases of Nut/OS and most people don't care much.

Some more background information: Waiting threads line up in priority ordered queues, so more than one thread may wait for the same event. Events are posted to a wait queue, moving the thread from waiting (sleeping) state to the ready-to-run state. A running thread may also broadcast an event to a specified queue, waking up all threads on that queue.

Usually a woken up thread takes over the CPU, if it's priority is equal or higher than the currently running thread. However, events can be posted asynchronously, in which case the posting thread continues to run. Interrupt routines must always post events asynchronously. Actually NutEventPostAsync is one of the rare API routines which can be called by interrupt routines.

Stream I/O

Typical C applications make use of the standard I/O library, which is provided by the avr-libc for GCC and by ImageCraft's libraries. However, only simple devices are supported. Therefore Nut/OS provides its own library nutcrt, which overrides the functions of the compiler's runtime library.

The list of available functions is quite impressive:

```
void clearerr(FILE * stream);
int fclose(FILE * stream);
void fcloseall(void);
FILE *_fdopen(int fd, CONST char *mode);
int feof(FILE * stream);
int ferror(FILE * stream);
int fflush(FILE * stream);
int fgetc(FILE * stream);
char *fgets(char *buffer, int count, FILE * stream);
int _fileno(FILE * stream);
void _flushall(void);
FILE *fopen(CONST char *name, CONST char *mode);
int fprintf(FILE * stream, CONST char *fmt, ...);
int fpurge(FILE * stream);
```

```
 int fputc(int c, FILE * stream);
 int fputs(CONST char *string, FILE * stream);
 size_t fread(void *buffer, size_t size, size_t count, FILE * stream);
 FILE *freopen(CONST char *name, CONST char *mode, FILE * stream);
 int fscanf(FILE * stream, CONST char *fmt, ...);
 int fseek(FILE * stream, long offset, int origin);
 long ftell(FILE * stream);
 size_t fwrite(CONST void *data, size_t size, size_t count, FILE *
stream);
 int getc(FILE * stream);
 int getchar(void);
 int kbhit(void);
 char *gets(char *buffer);
 int printf(CONST char *fmt, ...);
 int putc(int c, FILE * stream);
 int putchar(int c);
 int puts(CONST char *string);
 int scanf(CONST char *fmt, ...);
 int sprintf(char *buffer, CONST char *fmt, ...);
 int sscanf(CONST char *string, CONST char *fmt, ...);
 int ungetc(int c, FILE * stream);
 int vfprintf(FILE * stream, CONST char *fmt, va_list ap);
 int vfscanf(FILE * stream, CONST char *fmt, va_list ap);
 int vsprintf(char *buffer, CONST char *fmt, va_list ap);
 int vsscanf(CONST char *string, CONST char *fmt, va_list ap);
```

In addition, the following low level I/O functions are available:

```
 int _close(int fd);
 int _open(CONST char *name, int mode);
 int _read(int fd, void *buffer, size_t count);
 int _write(int fd, CONST void *buffer, size_t count);
 int _ioctl(int fd, int cmd, void *buffer);
 long _filelength(int fd);
```

This set of functions allows to port many existing PC applications without too much effort. Nevertheless, there is an important difference to standard C applications written for the PC. Typical embedded systems do not pre-define devices for stdin, stdout and stderr. Thus, the well know ‘Hello world’

```
#include <stdio.h>

int main(void)
{
    printf("Hello world!\n");
}
```

requires a few additional lines of code on Nut/OS.

```
#include <stdio.h>
#include <dev/usartavr.h>

int main(void)
{
    unsigned long baud = 115200;
    /* Register the device we want to use. */
    NutRegisterDevice(&devUsartAvr0, 0, 0);
    /* Assign the device to stdout. */
    freopen("uart0", "w", stdout);
    /* Optionally set the baudrate of the serial port. */
    _ioctl(_fileno(stdout), UART_SETSPEED, &baud);
```

```
    printf("Hello world!\n");
}
```

Btw., a typical pitfall is to mix Nut/OS and compiler libraries for standard C functions. If your application hangs in stdio functions or crashes when calling them, then check your linker map file to make sure, that these functions are loaded from Nut/OS libraries. If not, you may forgot to add nutcrt to the list of libraries.

File System

Neither Nut/OS nor Nut/Net require a file system, but Webservers are designed with a file system in mind. To make things easier for the programmer, Nut/OS provides a very simple file system named UROM, where files are located in code ROM. A special tool, crurom (create UROM) is used to convert directories into a C source file, which is then compiled and linked to your application code.

In addition, a preview release of a read-only FAT file systems is available, which requires additional hardware like an IDE or Compact Flash interface. This part is still under heavy development. Visit the project page on the Ethernut website and subscribe to the Ethernut mailing list for latest informations.

Device Drivers

Device drivers, wether written as Nut/OS extensions or as part of an application can define there own interrupt routines (check the compiler documentation) or may register callback routines by calling NutRegisterIrqHandler.

In opposite to desktop computers, tiny embedded systems do not provide a memory management hardware unit to protect memory areas or I/O port access. As a result, there is no special ‘kernel mode’ for device drivers to run in. Thus, there is no requirement for a device driver at all. To phrase it favorably, application code can freely use any kind of resource.

One advantage of device drivers is still left for Nut/OS: Abstraction. If a device driver exists for, let’s say, an LCD display, it may be used with stdio streams like other devices. Device usage can be switched easily by changing a few lines of code. Look at our ‘Hello world’ output going to an LCD.

```
#include <stdio.h>
#include <dev/hd44780.h>
#include <dev/term.h>

int main(void)
{
    /* Register the device we want to use. */
    NutRegisterDevice(&devLcd, 0, 0);
    /* Assign the device to stdout. */
    freopen("lcd", "w", stdout);

    printf("Hello world!\n");
}
```

Note, that real applications may contain a lot of input and output statements. Using stream I/O devices offers to write code, which is almost device independent. Provided, that Nut/OS drivers exist for the devices to be used.

Nut/Net

Network enabling, an overview.

Most TCP/IP implementations came from desktop PCs, requiring large code and buffer space. Available memory of embedded systems like the Ethernut board is much smaller. Nut/Net has been specifically designed for small systems.

Although this chapter tries to explain some basics, it makes no attempt to describe all aspects of TCP/IP in full detail. It is assumed that you have a working knowledge of the protocol.

Also check the Nut/OS API Reference for a detailed description.

Network Device Initialization With DHCP

Before using any Nut/Net function, the application must register the network device driver by calling NutRegisterDevice and configure the network interface by calling NutDhcpIfConfig or similar routines. NutDhcpIfConfig will try to retrieve the local IP address, network mask and default gateway from a DHCP server. If no DHCP server responds within a specified number of milliseconds, NutDhcpIfConfig uses the previously stored configuration from the on-chip EEPROM of the ATmega CPU. If the EEPROM doesn't contain any valid address, NutDhcpIfConfig will wait for an ICMP packet and use the IP address contained in its header. In this case the netmask will be 255.255.255.0 and no default gateway will be configured.

Applications may also choose to configure a fixed IP address and network mask by calling NutNetIfConfig or may even use both, NutDhcpIfConfig and NutNetIfConfig. To make things even more complicated, there are two IP addresses stored in EEPROM, a soft and a hard one. The soft IP address is the last used one. If Ethernut reboots and no DHCP server is available, then the soft IP address will be used again. If, however, a hard IP address is stored in EEPROM, NutDhcpIfConfig will not send out any DHCP query, but use the hard IP address immediately.

Even worse, DHCP requires an Ethernet (MAC) address to be assigned to the node first, which can be passed to NutDhcpIfConfig or may not be passed, in which case the DHCP client expects this address in EEPROM. The following table lists all Nut/Net specific item stored in the EEPROM.

Name	Type	Default	Description
size	Byte value	31	Total size of the configuration structure. Used to check validity.
name	Character array	eth0	Name of the network interface. Maximum size is 8 characters.
mac	Byte array	000698000000	6 bytes unique MAC address.
ip_addr	IP address	0.0.0.0	Last used IP address.
ip_mask	IP address	255.255.255.0	Configured IP mask.
gateway	IP address	0.0.0.0	Default gateway IP.
cip_addr	IP address	0.0.0.0	Configured IP address.

A certain part of code has been proven to fulfill most requirements:

```
if (NutDhcpIfConfig("eth0", 0, 60000)) {
  u_char my_mac[] = { 0x00, 0x06, 0x98, 0x20, 0x00, 0x00 };
  if (NutDhcpIfConfig("eth0", my_mac, 60000)) {
    u_long ip_addr = inet_addr("192.168.192.100");
    u_long ip_mask = inet_addr("255.255.255.0");
    NutNetIfConfig("eth0", my_mac, ip_addr, ip_mask);
  }
}
```

In the first call to NutDhcpIfConfig no MAC address is passed. If one is found in EEPROM, then the DHCP server will be queried for the related IP settings and the function returns 0. Otherwise it returns -1, but we do not know, wether the EEPROM doesn't contain a valid MAC address or DHCP failed. Thus NutDhcpIfConfig is called once again, but this time with a hard coded MAC address. If it fails again, then we are sure, that there is no valid EEPROM contents and no available DHCP server. Last thing we can do is to pass a hard coded IP address and IP mask to NutNetIfConfig to get the network interface up and running.

NutNetIfConfig will store the specified address as a hard IP address into the EEPROM. Remember, that DHCP will check the hard IP address. When Ethernut reboots again and call NutDhcpIfConfig, it will use this hard IP address and never try DHCP again unless the EEPROM is erased.

Why is this so complicated? Well, it isn't really, but granted, it isn't put straight either. Since its early releases, this part had been changed often and confused newcomers even more often. There are two things to consider. Nowadays most networks use DHCP and when Ethernut initially boots with DHCP, it will reboot without problems later. Second, setting up IP addresses is beyond the scope of an operating system. Instead, this is part of the user interface and therefore part of the application. From the application's point of view, Nut/Net network configuration is quite flexible.

Socket API

On top of the protocol stack Nut/Net provides an easy to use Application Programming Interface (API) based on sockets. A socket can be thought of as a plug socket, where applications can be attached to in order to transfer data between them. Two items are used to establish a connection between applications, the IP address to determine the host to connect to and a port number to determine the specific application on that host.

Because Nut/Net is specifically designed for low end embedded systems, its socket API is a subset of what is typically available on desktop computers and differs in some aspects from the standard Berkeley interface. However, programmers used to develop TCP/IP applications for desktop system will soon become familiar with the Nut/Net socket API.

TCP/IP applications take over one of two possible roles, the server or the client role. Servers use a specific port number, on which they listen for connection requests. The port number of clients are automatically selected by Nut/Net.

Nut/Net provides a socket API for the TCP protocol as well as the UDP protocol. The first step to be done is to create a socket by calling NutTcpCreateSocket or NutUdpCreateSocket.

TCP server applications will then call NutTcpAccept with a specific port number. This call will block until a TCP client application tries to connect that port number. After a connection has been established, both partners exchange data by calling NutTcpSend and NutTcpReceive.

A simple TCP client looks like this:

```
TCPSOCKET *sock = NutTcpCreateSocket();
NutTcpConnect(sock, inet_addr("192.168.192.2"), 80);
NutTcpSend(sock, "Hello\r\n", 7);
NutTcpReceive(sock, buff, sizeof(buff));
NutTcpCloseSocket(sock);
```

A Nut/Net TCP server looks similar:

```
TCPSOCKET *sock = NutTcpCreateSocket();
NutTcpAccept(sock, 80)
NutTcpReceive(sock, buff, sizeof(buff));
NutTcpSend(sock, "2U2\r\n", 5);
NutTcpCloseSocket(sock);
```

Furthermore, Nut/OS allows to attach TCP sockets to standard I/O streams, which makes our code looking more familiar, at least for Linux people. The following line is used to attach a connected socket to a Nut/OS stream.

```
stream = _fdopen((int)((uptr_t) sock), "r+b");
```

The code can use fprintf, fputs, fscanf, fgets and other stdio functions to talk TCP.

UDP server applications will provide their port number when calling NutUdpCreateSocket, while UDP client applications pass a zero to this call, in which case Nut/Net selects a port number greater than 1023, which is currently not in use. Data is transferred by calling NutUdpSendTo and NutUdpReceiveFrom, quite similar to the well known BSD functions sendto() and recvfrom(). NutUdpDestroySocket may be called to release all memory occupied by the UDP socket structure.

While UDP read provides a timeout parameter, TCP read doesn't. At least not directly, but a some special ioctrls are available, including one to set the read timeout. BSD calls them socket options, and so does Nut/Net.

```
unsigned long to = 1000; /* millisecs */
NutTcpSetSockOpt(sock, SO_RCVTIMEO, &to, sizeof(to));
```

Like with most TCP/IP stacks, there is no provision to set any connection timeout value and NutTcpConnect() may block for half a minute or more. That's how TCP/IP is designed. Most people experience this from the webbrowser on the desktop PC, when trying to connect a webserver that doesn't respond. Nevertheless, this seems to be unacceptable for embedded systems and may be changed in later Nut/Net releases.

Another socket option allows to modify the TCP maximum segment size.

```
unsigned long to = 1024;
NutTcpSetSockOpt(sock, TCP_MAXSEG, &mss, sizeof(mss));
```

It is also possible to define an initial TCP window size.

```
unsigned long win = 8192;
NutTcpSetSockOpt(sock, SO_RCVBUF, &win, sizeof(win));
```

The maximum segment size (MSS) and the initial window size may become useful when you need the maximum TCP throughput. The 8-bit AVR CPU running at 14.7456 MHz on the Ethernut easily reaches 2 MBit with the default values.

One last note to an often asked question. The number of concurrent connections is limited by memory space only. It is generally no problem to run several TCP and/or UDP server and/or client connections on a single Ethernut. The application may even use the same socket in more than one thread.

HTTP

The Hypertext Transfer Protocol (HTTP) is based on TCP. Nut/Net offers a set of helper APIs to simplify writing Embedded Webservers, which are described in detail in the Nut/OS API Reference.

TCP

The Transmission Control Protocol (TCP) is a connection oriented protocol for reliable data transmission. Nut/Net takes care, that all data is transmitted reliable and in correct order. On the other hand this protocol requires more code and buffer space than any other part of Nut/Net.

Applications should use the socket API to make use of the TCP protocol.

UDP

The advantage of the User Datagram Protocol (UDP) is its reduced overhead. User data is encapsulated in only eight additional header bytes and needs not to be buffered for retransmission. However, if telegrams get lost during transmission, the application itself is responsible for recovery. Note also, that in complex networks like the Internet, packets may not arrive in the same order as they have been sent.

Applications should use the socket API to make use of the UDP protocol.

ICMP

The Internet Control Message Protocol.

Nut/Net automatically responds to an ICMP echo request with an ICMP echo reply, which is useful when testing network connections with a Packet InterNet Groper (PING) program, which is available on nearly all TCP/IP implementations for desktop computers.

IP

The Internet Protocol is used by UDP and TCP. Typical applications do not directly use this network layer.

ARP

The Address Resolution Protocol is used by IP over Ethernet to resolve IP and MAC address relations.

Ethernet Protocol

This physical network layer is used by IP, ICMP and ARP.

PPP

The Point to Point is as an alternative to the Ethernet protocol and can be used with serial ports, modems, GPRS etc.

Nut/Net provides PPP client mode only. That means, other nodes can be actively connected, but Nut/Net can't listen to incoming connection attempts. This is related to establishing PPP connections and should not be confused with TCP/UDP client and server capabilities, which are fully available over PPP.

802.11 WLAN

Working with PCMCIA and CF cards, but experimental.

Bluetooth

Seems to be working too, but not included. Visit

http://btnode.ethz.ch/

for more information.

Conversion Functions

If multi-byte values are to be transferred over the network in binary form, the most significant byte must always appear first, followed by less significant bytes. This is called the network byte order, which differs from the host byte order, the order how multi-byte values are stored in memory. The AVR compilers store least significant bytes first. Several functions are provided to swap bytes from network byte order to host byte order or vice versa.

htonl() and htons() convert 4-byte resp. 2-byte values from host to network byte order, while ntohl() and ntohs() convert 4-byte resp. 2-byte values from network to host byte order.

More conversions are provided by inet_addr() and inet_ntoa(). While the first converts an IP address from the decimal dotted ASCII representation to a 32-bit numeric value in network byte order, the second routine offers the reverse function.

Network Buffers

Nut/Net uses a special internal representation of TCP/IP packets, which is designed for minimal memory allocation and minimal copying when packets are passed between layers. Application programmers don't need to care about this internal stuff. It is included here in case you might become interested in looking to the Nut/Net source code. Network buffers are one of the central data structure within Nut/Net.

A network buffer structure contains four equal substructures, each of which contains a pointer to a data buffer and the length of that buffer. Each substructure is associated to a specific protocol layer, datalink, network, transport and application layer. An additional flag field in the network buffer structure indicates, if the associated buffer has been dynamically allocated.

Network buffers are created and extended by calling NutNetBufAlloc and destroyed by calling NutNetBufFree. When a new packet arrives at the network interface, the driver creates a network buffer with all data stored in the datalink substructure. The Ethernet layer will then split this buffer by simply setting the pointer of the network buffer substructure beyond the Ethernet header and

adjusting the buffer lengths before passing the packet to the IP or ARP layer. This procedure is repeated by each layer and avoids copying data between buffers by simple pointer arithmetic.

When application data is passed from the top layer down to the driver, each layer allocates and fills only its specific part of the network buffer, leaving buffers of upper layers untouched. There is no need to move a single data byte of an upper layer to put a lower level header in front of it.

Simple TCP Server

Walking through a typical Nut/Net application.

This chapter explains how to use Nut/OS and Nut/Net to create a simple TCP server program.

Initializing the Ethernet Device

As with other Nut/OS applications you need to declare a function named main, containing an endless loop. The loop should call NutSleep() or any other blocking function to enable sibling threads to take over the CPU.

```
#include <sys/timer.h>
int main(void)
{
    for(;;) {
        NutSleep(500);
    }
}
```

To communicate via Ethernet, you have to initialize the Ethernet hardware. This takes two steps. The first is to register the device. A call to NutRegisterDevice will add all hardware dependent routines and data structures to your final code. The I/O port address and interrupt number parameters of this call are left for historical reasons and ignored by most device drivers.

```
NutRegisterDevice(&devEth0, 0, 0);
```

devEth0 is the device information structure of the LAN device driver. It contains the device name (eth0), the type of this interface (IFTYP_NET) and, among other things, the address of the hardware initialization routine. NutRegisterDevice will set up some data structures and initialize the controller hardware.

Network devices require specific configurations, which is done by calling

```
NutNetIfConfig("eth0", mac, 0, 0);
```

The first parameter is the name of the registered device. The second parameter needs some additional attention. It's an array of 6 bytes, containing the MAC address of the Ethernet controller. A MAC address, also referred to as the hardware or Ethernet address is a unique number assigned to every Ethernet node. The upper 24 bits are the manufacturer's ID, assigned by the IEEE Standards Office. The ID of Ethernut boards manufactured by egnite Software GmbH is 000698 hex. The lower 24 bits are the board's unique ID assigned by the manufacturer of the board.

Nut/Net will store this address in EEPROM, but we may also define a static variable to keep it in the application:

```
static u_char mac[] = { 0x00,0x06,0x98,0x09,0x09,0x09 };
```

The two remaining parameters of NutNetIfConfig are used to specify the minimum IP (Internet Protocol) information, the IP address of our node and the network mask. In our example we simply set both to zero, which will invoke a DHCP client to query this information from a DHCP server in the local network.

Therefore it may take some seconds until the call returns, depending on the response time of the DHCP server.

If there's no DHCP server available in your network, you must specify these two 4-byte values in network byte order. In this byte order the most significant byte is stored at the first address of a multi byte value, which differs from the byte order used by AVR processors (our host byte order). Fortunately Nut/Net provides a routine named inet_addr, which takes a string containing the dotted decimal form of an IP address and returns the required 4-byte value in host byte order.

If you want to assign IP address 192.168.171.2 with a network mask of 255.255.255.0, call:

```
NutNetIfConfig("eth0", mac, inet_addr("192.168.171.2"),
inet_addr("255.255.255.0"));
```

NutNetIfConfig will initialize the Ethernet controller hardware and should lit the link LED on your board, if it is properly connected to an Ethernet hub or switch. At this point Ethernut will already respond to ARP and ping requests:

```
#include <dev/nicrtl.h>
#include <sys/timer.h>
#include <arpa/inet.h>

static u_char mac[] = { 0x00,0x06,0x98,0x09,0x09,0x09 };

int main(void)
{
    NutRegisterDevice(&devEth0, 0x8300, 5);
    NutNetIfConfig("eth0", mac, 0, 0);
    for(;;) {
        NutSleep(500);
    }
}
```

Connecting a Client With a Server

Up to now we followed the standard path, common to all TCP/IP applications created for Ethernut. The next task is to create the application specific part.

In order to communicate via TCP, we need a TCP socket, which is actually a data structure containing all information about the state of a connection between two applications.

```
TCPSOCKET *sock;

sock = NutTcpCreateSocket();
```

This call allocates the data structure from heap memory, initializes it and returns a pointer to the structure. The next step specifies, wether we take over the client or the server part of the communication. As a client we would try to connect to a specified host on a specified port number. Here's an example to connect to port 12345 of the host with the IP address of 192.168.171.1:

```
NutTcpConnect(sock, inet_addr("192.168.171.1"), 12345);
```

In our sample application we decided to take over the server part, which is done by calling

```
NutTcpAccept(sock, 12345);
```

This call will block until a client tries to connect us. As soon as that happens, we can send data to the client by calling NutTcpSend or receive data from the client through NutTcpReceive.

Communicating with the Client

Most application protocols in the Internet environment exchange information by transmitting ASCII text lines terminated by a carriage return followed by a line feed character. This might not be a big problem while sending data, but it requires some extra effort for incoming data, as arriving segments may contain either a fraction of a line or multiple lines or both. And even sending data becomes more complicated with numeric values, because we need to transfer them to there ASCII representation first.

For stream devices Nut/OS offers the standard I/O stream library functions. In order to use them, you need to attach a connected TCP socket to a stream.

```
FILE *stream;

stream = _fdopen((int)sock, "r+b");
```

This device can be used like any other Nut/OS stream device and simplifies formatted and line oriented data I/O.

The first thing servers usually do after a client connected them is to send a welcome message:

```
fprintf(stream, "200 Welcome to Ethernut\r\n");
```

Note, that it's a good idea to prepend a numeric code in front of server responses. This way both, a human user as well as a client program can easily interpret the message. On the other hand, the message above occupies 26 bytes of SRAM space. Alternatives are:

```
fputs("200 OK\r\n", stream);
```

or

```
fputs_P(PSTR("200 Welcome to Ethernut\r\n"), stream);
```

In the second example the message is stored in program space and only temporarily copied to SRAM when needed. The PSTR() macro is only available with AVR-GCC. When using ICCAVR, you have to write it in a different way:

```
prog_char vbanner_P[] = "200 Welcome to Ethernut\r\n";
fputs_P(vbanner_P, stream);
```

In the next step TCP servers typically await a command from the client, perform the associated activity, return a response to the client and await the next command.

Disconnecting

Finally the client will disconnect or, as preferred, send a special command to force the server to disconnect. The server will then call

```
fclose(stream);
```

to release all memory occupied by the stream device and then call

```
NutTcpCloseSocket(sock);
```

to terminate the socket connection. Next, the server may create a new socket and wait for a new client connection.

Trying the Sample Code

The complete source code of a TCP server example can be found in subdirectory app/tcps of your installation directory. If there's no DHCP server in your local network, you need to modify the call to NutNetIfConfig in the C source file named tcps.c as explained above.

Note, that your PC and the Ethernut board should use the same network mask but different IP addresses. And they must reside in the same network, unless you add specific routes to the Nut/Net routing table. However, IP routing might get rather complex and is beyond the scope of this manual. You might refer to a good book explaining that matter.

Most local networks are configured as class C, which means, that a maximum of 254 different IP addresses are available and the IP network mask is specified as 255.255.255.0. All hosts in this network must have equal numbers in the first three parts of their IP addresses. In this case 192.168.171.1 and 192.168.171.5 belong to the same network, but 192.168.171.1 and 192.168.181.5 don't.

After you have done the changes, open a Linux console or DOS prompt with the GCC environment and enter

```
make
```

This will create an updated binary file named tcps.hex.

If your local network supports DHCP, you may use the precompiled binary for a first try, but may later modify the default MAC address.

After programming your Ethernut board with this binary, open a Linux console or DOS prompt window and type

```
telnet x.x.x.x 12345
```

replacing x.x.x.x with the IP address of your Ethernut board. The telnet window should display

```
200 Welcome to tcps. Type help to get help.
```

You may now enter any of the following commands using lower case letters:

```
memory         Displays the number of available SRAM bytes.
threads        Lists all created threads.
timers         Lists all running timers.
```

Reference Material

Interested in more?

Books

Comer D. Internetworking with TCP/IP, Vol I: Principles, Protocols, and Architecture, Prentice Hall

Covers many protocols, including IP, UDP, TCP, and gateway protocols. It also includes discussions of higher level protocols such as FTP, TELNET and NFS.

Comer D., Stevens D. Internetworking with TCP/IP, Vol II: Design, Implementation and Internals, Prentice Hall

Discusses the implementation of the protocols with many code examples.

Comer D., Stevens D. Internetworking with TCP/IP, Vol III: Client-Server Programming and Applications, Prentice Hall

Discusses application programming using the internet protocols. It includes examples of telnet, ftp clients and servers.

Stevens W. TCP/IP Illustrated Vol 1, Addison-Wesley

One of if not the most recommended introduction to the entire TCP/IP protocol suite, covering all the major protocols and several important applications.

Stevens W. TCP/IP Illustrated Vol 2, Addison-Wesley

Discusses the internals of TCP/IP based on the Net/2 release of the Berkeley System.

Stevens W. TCP/IP Illustrated Vol 3, Addison-Wesley

Covers some special topics of TCP/IP.

RFCs

RFCs (Request For Comment) are documents that define the protocols used in the Internet. Some are standards, others are suggestions or even jokes. Many Internet sites offer them for download via http or ftp.

Postel, Jon, RFC768: User Datagram Protocol

Postel, Jon, RFC791: Internet Protocol

Postel, Jon, RFC792: Internet Control Message Protocol

Postel, Jon, RFC793: Transmission Control Protocol

Plummer, D.C, RFC826: Ethernet Address Resolution Protocol

Braden, R.T, RFC1122: Requirements for Internet Hosts - Communication Layers

T. Berners-Lee, R. Fielding, and H. Frystyk, RFC1945: Hypertext Transfer Protocol

Web Links

http://www.ethernut.de Ethernut support

http://www.egnite.de Ethernut Online Shop

http://www.atmel.com Manufacturers of AVR microcontrollers

http://www.avrfreaks.org Lots of useful infos about AVR microcontrollers

Troubleshooting

Problem	Solution
The Configurator's main window is empty. There's no component tree shown.	Make sure to set the correct path to the repository on the first settings page. Reload the configuration file or restart the Configurator.
Building Nut/OS for platform XYZ fails with errors. Using the old method, building it directly in the source tree, fails too.	The configuration had been tested with different settings. However, some configurations may still fail. Standard builds should be done with the Configurator, specifying a separate build directory. (At the time of this writing 'standard' refers to configurations that worked with Nut/OS 3.4.)
I can't figure out, how to solve my problem.	The Ethernut mailing list (preferably the English list en-nut-discussion) is the recommended forum for Nut/OS users. Please be aware, that all the helpful people are volunteers. Try to be as friendly as possible and post Nut/OS related messages only. Do not ask questions about general programming or networking. There are be better places to get answers in such cases.

egnite Software GmbH
Westring 303
44629 Herne
Germany

Phone +49 (0)23 23-92 53 75
Fax +49 (0)23 23-92 53 74

Email info@egnite.de

http://www.egnite.de
http://www.ethernut.de